НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
«___»_____2025 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

на тему: «Програмний модуль для динамічного масштабування OpenStack на основі моніторингу навантаження»

Виконав:
студент IV курсу, групи ТВ-12
Онопрієнко Дмитро Олегович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник:
асистент Гейко О.О. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:
_____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
«___»_____2025р.

ЗАВДАННЯ

на дипломну роботу студенту

Онопрієнку Дмитру Олеговичу
(прізвище, ім'я, по батькові)

1. Тема роботи: Програмний модуль для динамічного масштабування OpenStack-на основі моніторингу навантаження
керівник роботи асистент Гейко Олег Олександрович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «02» червня 2025р. №1875-с
2. Строк подання студентом роботи «09» червня 2025р.
3. Вихідні дані до роботи: мова програмування Kotlin, фреймворк Spring, Gitlab, середовище IntelliJ IDEA, Prometheus
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): розробити програмний модуль для динамічного масштабування кафедрального OpenStack; інтегрувати модуль в інститутську інфраструктуру.

5. Перелік ілюстративного матеріалу: UML діаграма схеми взаємодії модулів системи, результати роботи програми, приклади налаштувань, схеми взаємодії додатку з системою моніторингу, діаграма класів.

6. Дата видачі завдання «31» жовтня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	30.10.2024	
2	Дослідження предметної області	31.10.2024 – 01.11.2024	
3	Дослідження існуючих рішень	02.11.2024 – 01.12.2024	
4	Постановка вимог до проєктування системи	02.12.2024 – 12.01.2025	
5	Розробка програмного продукту	13.01.2025 – 11.05.2025	
6	Тестування	12.05.2025 – 15.05.2025	
7	Захист програмного продукту	12.05.2025 – 15.05.2025	
8	Оформлення дипломної роботи	19.05.2025 – 01.06.2025	
9	Передзахист	02.06.2025 – 06.06.2025	
10	Захист	16.06.2025 – 27.06.2025	

Студент _____ Дмитро ОНОПРІЄНКО
(підпис) (ім’я, прізвище)

Керівник роботи _____ Олег ГЕЙКО
(підпис) (ім’я, прізвище)

РЕФЕРАТ

Метою даної роботи є автоматизація процесів управління ресурсами кластеру, що дозволить запобігати критичним навантаженням, зменшити залежність від ручних операцій та підвищити стабільність системи.

Результатом роботи став програмний модуль для динамічного масштабування кафедральної інфраструктури OpenStack, що дозволило вирішити проблему автоматичної реакції на аномальні зміни в життєвих показниках серверу.

Робота містить 63 сторінки, 11 рисунків, 2 додатки та 3 посилання.

Ключові слова: динамічне масштабування, OpenStack, Spring, Java, Kotlin, інфраструктура, моніторинг навантаження.

ABSTRACT

The purpose of this work is to automate cluster resource management processes, which will prevent critical loads, reduce dependence on manual operations, and increase system stability.

The result of the work was a software module for dynamic scaling of the departmental OpenStack infrastructure, which allowed solving the problem of automatic response to abnormal changes in server performance.

The work contains 63 pages, 11 figures, 2 appendices and 3 references.

Keywords: dynamic scaling, OpenStack, Spring, Java, Kotlin, infrastructure, load monitoring.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

СМ	- система моніторингу
КБІ	- конвеєр безперервної інтеграції
АК	- адміністратор кластеру
IDEA	- (англ. IntelliJ IDEA) інтегрована середа розробки, спеціалізована для мов програмування, заснованих на JVM.
Terraform	- відкрите програмне забезпечення для конфігурації обчислювальних ресурсів за моделлю "інфраструктура як код".
Prometheus	- система моніторингу та аналізу життєвих показників додатків та серверів з відкритим початковим кодом.

ВСТУП

У сучасному світі дуже часто в основі багатьох систем лежать хмарні обчислення, що потребують складної інфраструктури та її постійної підтримки для функціонування комплексних cloud систем. Хостинг серверних та веб додатків потрібен майже всюди, в тому числі для навчання студентів та підтримання працездатності навчальних закладів. В університетах вони використовуються для вирішення широкого кола задач, в тому числі наступні:

- Збереження інформації про студентів;
- Ведення реєстру оцінок та атестацій;
- Зберігання даних про обрані предмети на семестр;
- Управління обраними дипломними та курсовими темами
- Архівування зданих робіт.

Через необхідність в такому менеджменті даних, що є критичними для роботи університету, кафедра створила особистий кластер для хмарних обчислень та збереження інформації на основі існуючих апаратних засобів та рішень з відкритим початковим кодом з метою об’єднання наявних комп’ютерів між собою та створення на базі цього cloud інфраструктури для задоволення потреб навчального закладу.

Ця система потребує великої кількості менеджменту та підтримки її для коректного функціонування, а також упередження збоїв та швидкого відновлення в разі критичної поведінки. Для цього необхідно постійно відслідковувати стан кафедрального кластеру, навантаження на нього та прийняття рішень на основі отриманих даних. Автоматизація цього процесу дуже важлива, оскільки мануальні операції над інфраструктурою доволі важкі та потребують постійної залученості людини, що має спеціальні навички для цього. До того ж, час реакції може складати до декількох годин, оскільки моніторинг роботи кластера в такому разі проводився б раз в деякий період часу, через що можливі ситуації, коли такі маніпуляції не проводилися б вчасно.

Мета цієї роботи складається в автоматизації масштабування хмарного кластеру кафедри та запобігання відмов системи через надмірне навантаження. Це дасть можливість не проводити складні інфраструктурні маніпуляції вручну і не буде потребувати участі досвідченого спеціаліста в масштабуванні кластеру. В той же час це зменшить час реагування на аномальну поведінку та надмірне навантаження на існуючі вузли за допомогою автоматизованого відслідковування життєвих показників хмари. Такий підхід дасть можливість автоматично створювати додаткові віртуальні машини для нормалізації роботи кластеру та їх виведення з нього в разі зменшення навантаження.

1 ПОСТАНОВКА ЗАДАЧІ

Метою даної роботи є реалізація автоматизації масштабування існуючої хмарної інфраструктури кафедри в разі надмірного навантаження. В режимі реального часу мають оброблятись дані про життєві показники серверів і в разі аномальної поведінки, а саме – збільшення завантаженості віртуальної машини зверх певного порогу, мають відбутись автоматичні дії, що піднімуть додаткові вузли в кластері для більш рівномірних та очікуваних показників і упередження відмов системи. В цей же час, в разі зменшення навантаження та стабілізації роботи системи, має відпрацювати зворотний механізм, що прибере надлишкові вузли для мінімізації затрат системи та більш якісної взаємодії.

Така система буде реалізована за допомогою інтеграції трьох модулів інфраструктури – системи моніторингу (СМ), модуля, що відповідає за конфігурацію інфраструктури, та спеціального додатку, що буде поєднувати між собою дві названі вище підсистеми.

Перший модуль відповідає за збір метрик та моніторинг даних з хмарного кластера кафедри, їх аналіз та відправку налаштованих сповіщень. Цей модуль конфігурується таким чином, що він бере вибірку даних за певний період, обчислює за встановленою попередньо формулою результат, який має вигляд булевого значення (true або false), і на основі цього результату, якщо він був позитивним, відправляє сповіщення за допомогою завчасно налаштованого способу, яким може бути повідомлення в месенджер, на пошту, HTTP Webhook тощо. Це допоможе реалізувати автоматичну реакцію на зміни в поведінці вузлів.

Другий модуль відповідає за налаштування та параметри, що задають інфраструктуру. Він чітко описує конфігурації хмари, взаємодію між вузлами, їх відповідність одне одному, відповідає за налаштування внутрішніх мереж та інтеграцію з глобальною мережею. Завдяки змінам в цьому модулі конфігурацій можна масштабувати потрібні ресурси, як мануально, за допомогою ручних маніпуляцій, так і автоматично, попередньо налаштувавши його для цього.

Третій модуль виступає в якості посередника, що дозволяє з легкістю інтегрувати між собою дві підсистеми, що описані вище. Він збирає дані про зміни в навантажені, що лежать в модулі відповідальному за моніторинг кластеру, аналізує їх, та в разі потреби вносить зміни в конфігурації ресурсів інфраструктури та застосовує їх. В разі попереднього масштабування і стабілізації метрик в подальшому, система має повертати попередню кількість робочих вузлів кластеру.

Для реалізації задачі необхідно:

1. Розробити серверний додаток, що відповідає за прийняття рішень про масштабування;
2. Інтегрувати CM з додатком;
3. Налаштувати правила повідомлень про аномалії в життєвих показниках;
4. Сконфігурувати взаємодію додатку з Terraform через Gitlab CI;
5. Розробити CI Pipeline, що буде змінювати налаштування інфраструктури по запиту.

Також під час розробки системи треба врахувати необхідність реагування модулю динамічного масштабування лише на довгострокові підвищення навантаження, оскільки додавання нових вузлів та обчислювальних потужностей в інфраструктуру може займати великий проміжок часу та це забирає досить велику кількість ресурсів, через що відпрацювання на короткострокові аномалії є не раціональним.

Висновки до розділу 1

У даному розділі було сформульовано основні цілі та завдання дипломної роботи, яка полягає в автоматизації масштабування хмарної інфраструктури кафедри у разі надмірного навантаження. Головною метою є забезпечення динамічного підвищення доступності системи шляхом автоматичного додавання

вузлів при зростанні навантаження та їх відключення при стабілізації роботи, що дозволить оптимізувати витрати на обчислювальні ресурси.

2 ОПИС ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Система автоматичного масштабування є комплексним рішенням, що складається з декількох інтегрованих між собою програмних модулів та технологій, що взаємодіють між собою задля автоматизації процесу підтримки життєдіяльності хмарного кластеру кафедри навіть під час великих навантажень, спричинених різким одночасним напливом багатьох користувачів на ресурси єКафедри. До складу системи входять модулі збору інформації про життєві показники віртуальних машин, їх аналізатори та системи сповіщень про аномалії, попередньо налаштовані на певну реакцію, модуль конфігурації ресурсів хмари, що відповідає за усі внутрішні зміни, та додаток інтеграції систем сповіщень та провокування змін в параметрах інфраструктури.

2.1 Аналіз наявних рішень

Потреба в автоматизованому масштабуванні на базі моніторингу навантаження є не новою та є багато вже наявних рішень, що використовуються по всьому світу. Прикладами таких систем є Amazon (AWS) Auto Scaling Groups, Kubernetes Horizontal Pod Autoscaling тощо. Такі системи мають нативну інтеграцію з системами моніторингу та детально пропрацьовані правила для масштабування, як вгору так і вниз. Основна проблема з використанням таких систем – вони підходять лише для певних провайдерів інфраструктури або ж вони створені спеціально для одного конкретного набору технологій. Наприклад, Amazon Auto Scaling Groups розроблений саме для використання всередині екосистеми Amazon Web Services (AWS).

Для кластеру кафедри в якості СМ використовується Prometheus, а для створення серверної екосистеми було розгорнуто OpenStack, що є не дуже популярним рішенням для хмарного використання, через що виникають певні складнощі з інтеграцією сторонніх рішень, які розроблені під більш поширені

технології. Для конфігурації інфраструктури кафедрального кластеру використовується Terraform, що налаштовує її за моделлю «інфраструктура як код». Код Terraform з усіма необхідними конфігураціями зберігається в розгорнутому на базі обчислювальних потужностей кафедри Gitlab. Через сукупність всіх цих факторів, оптимальним рішенням є створення нового модуля для динамічного масштабування ресурсів кафедрі на основі моніторингу навантаження, що буде інтегрувати між собою ці системи та буде оптимальним для потреб єКафедри.

2.2 Проблематика інтеграції з системою моніторингу

Інтеграція з системою моніторингу може мати декілька варіантів реалізації, в кожної з котрих є свої переваги та недоліки. Один з можливих способів – постійне опитування через API системи моніторингу додатком, відповідальним за прийняття рішень щодо масштабування. Цей спосіб отримання даних називається «pull» модель (рис. 2.1). Така схема взаємодії надає можливість отримувати детальні вибірки даних за певний період часу, що дає можливість краще відстежувати зміни в життєвих показниках вузлів, більш детально їх аналізувати та бачити перепади цих показників. В такому підході ми отримуємо більший об’єм інформації про хмарний кластер, але виникає складність в самостійній обробці цих даних, оскільки для цього треба реалізувати окрему систему, що буде постійно проводити обчислення над отриманими показниками за декількома методами, що до того ж створює додаткове навантаження на кластер і потребує використання механізму планування задач, щоб коректно реалізувати “pull” модель взаємодії.

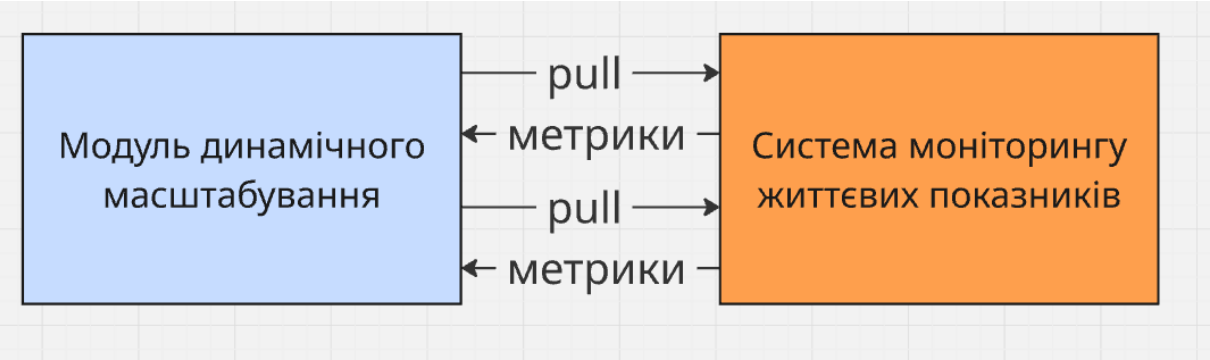


Рис. 2.1 – “pull” модель взаємодії

В той же час є другий спосіб, який базується на “push” моделі (рис. 2.2) й має таку перевагу, як більш проста обробка даних на стороні додатку, що відповідає за прийняття рішення про масштабування, та ініціативність зі сторони системи моніторингу. Тобто така система сама відслідковує попередньо налаштовані показники. Користувачі та розробники системи конфігурують всередині модулю моніторингу правила, за якими будуть оброблятись дані, і в разі спрацьовувань цих правил вона буде відсилати сповіщення на завчасно налаштованого отримувача, яким може бути бот в месенджері, електронна пошта, Webhook тощо. В такому способі взаємодії легше сконфігурувати обробку ряду даних за період часу, оскільки такі обчислення виконує сама моніторингова система за інтегрованими та протестованими її розробниками математичними моделями та формулами, що дозволяє збільшити точність отриманих результатів та запобігти помилок в обробці зібраних даних. До того ж в таких системах наявна можливість гнучко змінювати та додавати отримувачів для сповіщень. Наприклад, в використаному в кластері кафедри Prometheus та Alertmanager є багато вже наявних «з коробки» інтеграцій зі сторонніми отримувачами повідомлень, такі як Telegram, Email, Webhook і навіть WeChat. Окрім цього, розробники Prometheus надають посилання та приклади використання сторонніх перевірених інтеграцій, що дозволяють використовувати в якості отримувача Discord, Slack, Signal тощо. В такому випадку основна задача зводиться до обирання правильної формули та налаштування отримувача, що буде

відповідати точці входу на стороні додатку, який вже буде виконувати функцію застосування змін та буде чинником їх ініціалізації на стороні інфраструктури.

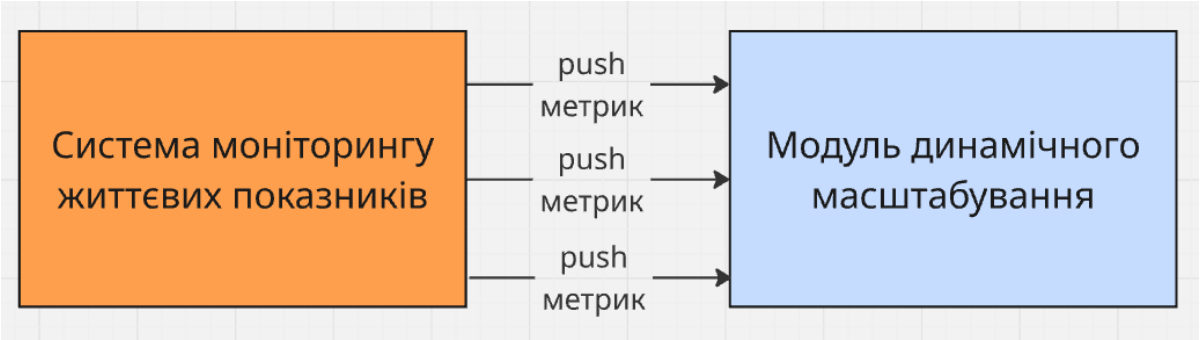


Рис. 2.2 – “push” модель взаємодії

2.3 Проблематика роботи з інфраструктурою

Через те, що технологія OpenStack, що використовується для розгортання кластеру, є проектом з відкритим початковим кодом та є доволі об’ємною, але не настільки популярною як аналогічні системи – виникає проблема із пошуком документації та статей, що пояснюють принципи роботи з інфраструктурою на базі цієї системи. Це ускладнює процес і розуміння взаємодії модулів хмари між собою.

Оскільки наша система має конфігуруватись за допомогою коду з використанням технології Terraform та зберігати дані про налаштування ресурсів у вигляді файлів в репозиторії, нам треба постійно підтримувати ці дані актуальними, оскільки цього вимагає парадигма «інфраструктура як код» (IaC). Це потребує реалізації додаткових інструментів для оновлення конфігурацій. Через це виникає проблема правильної інтеграції та підтримки стану, оскільки без цього ми не можемо розраховувати на те, що під час синхронізації інфраструктурних компонентів, наш кластер залишиться в тому ж стані, що був після внесених змін, оскільки «інфраструктура як код» буде вичитувати значення, що наявні в Git репозиторії з конфігурацією кластеру. Отже наше рішення вимагає додаткової

реалізації внесення автоматизованих змін в репозиторій налаштувань та їх збереження. В свою чергу, таке рішення потребує створення додаткових конвеєрів безперервної інтеграції (КБІ) і налаштування дозволів для них. Також окремої уваги потребує розробка коректної схеми взаємодії з системою безперервної інтеграції, оскільки ми маємо запускати цей конвеєр віддалено з нашого додатку, що відповідає за поєднання системи моніторингу та конфігурації інфраструктури, а також прийняття рішення про масштабування. Через це виникає додаткова задача безпеки, оскільки взаємодії з системами контролю версій та безперервної інтеграції вимагають токенів доступу, або інших способів автентифікації та підтвердження наявності можливостей для взаємодії та внесення змін у відповідні системи.

Висновки до розділу 2

У даному розділі було проведено детальний аналіз предметної області, пов’язаної з автоматичним масштабуванням хмарного кластеру кафедри. Розглянуто структуру системи, яка включає модулі збору даних, аналізу, сповіщень та управління ресурсами, що працюють у комплексі для підтримки стабільної роботи інфраструктури під час високих навантажень.

Проведено порівняльний аналіз існуючих рішень для автоматичного масштабування, таких як AWS Auto Scaling Groups та Kubernetes Horizontal Pod Autoscaling. Встановлено, що вони не підходять для специфічної інфраструктури кафедри, оскільки розроблені під конкретні екосистеми та не сумісні з використовуваними технологіями, зокрема OpenStack та Prometheus. Це обумовило необхідність розробки власного модуля, який би враховував особливості наявного середовища.

Окрему увагу приділено проблемам інтеграції з системою моніторингу, де порівняно дві моделі взаємодії – "pull" та "push". Визначено, що "push" модель є більш ефективною для даного випадку, оскільки дозволяє використовувати

вбудовані механізми обробки даних та сповіщень, що спрощує розробку та підвищує точність аналізу.

Також досліджено проблеми роботи з інфраструктурою, зокрема обмежену документацію для OpenStack та необхідність підтримки актуальності конфігурацій у рамках парадигми "інфраструктура як код". Виявлено, що для успішної реалізації системи потрібно розробити механізми автоматичного оновлення репозиторію та забезпечити безпеку взаємодії з системами контролю версій та CI/CD.

У цілому, проведений аналіз показав, що існуючі рішення не відповідають потребам кафедрального кластеру, що підтверджує доцільність розробки спеціалізованого модуля, який би інтегрував наявні технології та забезпечував ефективне масштабування ресурсів.

3 ЗАСОБИ РОЗРОБКИ

Важливою частиною кожного проекту є вибір стеку технологій, що будуть використані під час розробки. Оскільки існує багато різних мов програмування, фреймворків, бібліотек, то дуже важливо підібрати саме такі, що будуть краще підходити для конкретного проекту. Треба враховувати особливості кожної технології, для чого вона була розроблена, а також специфіку її використання. Також необхідно брати до уваги поріг входу до кожної з них, оскільки технологія може ідеально підходити для вирішення конкретної задачі, але її вивчення та опанування забере таку кількість ресурсів, що це буде не релевантно, навіть порівняно з інструментом, що потребує якихось додаткових дій але має легшу криву вивчення**.**

Також під час планування проектування необхідно враховувати вже наявні в системі технології та підлаштовуватись під них, перевикористовуючи наявні інструменти й розширюючи їх. Це спростить кодову базу системи загалом та її підтримку, а також зменшить необхідну кількість ресурсів для її функціонування та знизить поріг входу за знаннями та навичками для АК.

3.1 Мова програмування Kotlin

Kotlin — це статично типізована мова програмування, яка працює на віртуальній машині Java (JVM) [2]. Вона розроблена компанією JetBrains і набирає все більшої популярності, особливо для розробки мобільних додатків під Android та серверних додатків на Spring Framework. Kotlin є сучасною мовою, що пропонує багато функціональних можливостей, які спрощують написання коду, роблять його більш безпечним і легким для читання.

Основні переваги Kotlin полягають у його лаконічності та виразності. Завдяки його особливостям, таким як розширення функцій, null-безпека та smart casts, розробникам потрібно писати менше коду для досягнення тих самих

результатів, що призводить до підвищення продуктивності. Сумісність з Java є ще однією важливою перевагою: Kotlin може легко взаємодіяти з існуючим кодом Java, що полегшує перехід на Kotlin та використання вже розроблених бібліотек. Це дозволяє легше інтегруватись з системами, в яких є інструменти для розробки, що були створені для Java, оскільки в такому випадку їх можна використовувати нативно, без додаткових зусиль під час розробки модулів взаємодії зі сторонніми системами. Крім того, Kotlin вважається більш безпечною мовою порівняно з Java, оскільки він має вбудовані механізми захисту від типових помилок, таких як NullPointerException.

3.2 Spring Framework

Spring Framework — це популярний open-source фреймворк для розробки Java-додатків, який надає комплексні можливості для створення корпоративних рішень. Він включає модулі для роботи з базами даних, безпекою, веб-розробкою та іншими аспектами сучасних програм. Spring спрощує розробку завдяки принципам інверсії контролю (IoC) та dependency injection (DI), що дозволяє розробникам створювати гнучкі та легкі у підтримці додатки. Фреймворк також підтримує різні технології, такі як Hibernate, JDBC та REST API, що робить його універсальним інструментом і дає можливість легко створювати серверні додатки та налаштовувати їх. Саме простота розробки додатків із HTTP API дає змогу легко інтегрувати модуль автоматичного масштабування з іншими системами, що вимагають можливості робити запити та відправляти webhook.

Окрім цього, фреймворк надає зручні інструменти для використання Java та JVM мов програмування для використання сторонні HTTP API за допомогою вбудованих модулів, таких як RestTemplate, що дозволяє конструювати та виконувати HTTP запити будь якої складності прямо зсередини Spring додатку. Це спрощує роботу з мікросервісною архітектурою і інтеграцією зі сторонніми програмними інтерфейсами.

Одна з головних переваг Spring — це модульність, яка дозволяє використовувати лише ті компоненти, які необхідні для конкретного проекту. Spring Boot, частина екосистеми, ще більше спрощує розробку, автоматизуючи налаштування та конфігурацію. Фреймворк також забезпечує високу продуктивність завдяки оптимізованим механізмам кешування та асинхронній обробці. Крім того, Spring має велику спільноту та багато документації, що полегшує навчання та вирішення проблем.

Ще одна важлива перевага — інтеграція з іншими технологіями, такими як Spring Security для аутентифікації, Spring Data для роботи з базами даних та Spring Cloud для розподілених систем. Фреймворк підтримує різні стилі програмування, включаючи AOP (аспектно-орієнтоване програмування), що дозволяє краще керувати логікою додатків.

Завдяки своїй гнучкості та масштабованості Spring став стандартом для розробки корпоративних й не тільки Java-додатків. Він також постійно оновлюється, що забезпечує підтримку сучасних трендів у розробці програмного забезпечення.

3.3 Prometheus

Prometheus — це система моніторингу та сповіщення з відкритим кодом, розроблена для надійного збору та аналізу метрик у реальному часі. Вона спеціалізується на моніторингу динамічних середовищ, таких як контейнери та мікросервіси, і використовує часові ряди для зберігання даних. Prometheus працює за принципом "pull"-моделі, періодично запитуючи метрики з цільових сервісів через HTTP.

Одна з головних переваг — простота інтеграції: Prometheus підтримує різноманітні експортери для збору даних із різних джерел, включаючи бази даних, вебсервери та пристрої інтернету речей. Він також надає власну потужну мову

запитів (PromQL), яка дозволяє гнучко аналізувати метрики та будувати складні алерти.

Окрім цього, він має вже розроблені, інтегровані та протестовані математичні моделі, що аналізують отримані з сервісів дані про життєві показники, вираховують результати для правил про сповіщення та проводять обчислення над часовими рядами значень метрик. Ці моделі постійно підтримуються та тестуються, що робить цей механізм більш надійним і точним. Це в свою чергу запобігає некоректним відпрацюванням та збільшує точність отриманих результатів.

Prometheus особливо популярний у DevOps-середовищах завдяки своїй масштабованості та сумісності з Kubernetes та іншими оркестраторами. Він не залежить від зовнішніх сервісів, що робить його ідеальним для локальних та хмарних розгортань. Його активна спільнота та широкий вибір інтеграцій роблять Prometheus одним із найкращих інструментів для моніторингу сучасних інфраструктур.

3.4 Prometheus Alertmanager

Alertmanager — це компонент екосистеми Prometheus, який відповідає за обробку, групування та відправку сповіщень на основі алертів, створених у Prometheus. Він дозволяє фільтрувати дублікати, групувати пов’язані сповіщення та направляти їх до правильних каналів, таких як email, Slack або webhook. Завдяки цьому адміністратори отримують лише релевантні сповіщення, що допомагає уникнути "шторму" непотрібних повідомлень.

Одна з ключових переваг — гнучке налаштування маршрутизації сповіщень, що дозволяє налаштовувати різні політики для різних типів алертів та команд. Alertmanager також підтримує придушення (silencing) сповіщень, що корисно під час планового технічного обслуговування. Крім того, він інтегрується з різними системами спілкування, що робить його універсальним інструментом для організації ефективного моніторингу та дозволяє й оперативно реагувати на

аномалії в життєвих показниках завдяки швидким сповіщенням через налаштованих отримувачів, серед яких можуть бути Telegram, Discord, Signal тощо.

3.5 Git

Git — це розподілена система контролю версій, яка дозволяє відстежувати зміни у файлах та співпрацювати над проектами [5]. Вона дає змогу фіксувати різні стани коду, повертатися до попередніх версій та паралельно працювати над різними функціями через гілки. Git є одним із найпопулярніших інструментів у розробці ПЗ завдяки своїй гнучкості та потужним можливостям.

Основні переваги Git включають швидкість, надійність та підтримку розподіленої роботи. Кожен розробник має повну копію репозиторію, що дозволяє працювати офлайн та зменшує ризик втрати даних. Крім того, Git інтегрується з різними хостинговими сервісами, як GitHub чи GitLab, спрощуючи спільну роботу над проектами.

3.6 GitLab

GitLab — це платформа для керування репозиторіями Git, яка надає інструменти для спільної розробки, відстеження задач, code review та управління проектами. Вона поєднує функції системи контролю версій (VCS) з можливостями DevOps, що дозволяє командам ефективно працювати над кодом у єдиному середовищі. GitLab підтримує як хмарні, так і локальні (self-hosted) рішення, що робить його гнучким інструментом для різних потреб.

Одним з найважливіших інструментів GitLab для DevOps є GitLab CI. GitLab CI (Continuous Integration) — це вбудований інструмент GitLab для автоматизації збірки, тестування та розгортання коду. Він дозволяє налаштувати конвеєри, які автоматично виконують певні дії при зміні коду, забезпечуючи швидку інтеграцію змін і якість продукту. Також він має спеціальний прикладний програмний

інтерфейс (API) для запуску конвеєрів та виконання в них певних дій, в тому числі зміну початкового коду в репозиторії та виконання консольних команд будь якої складності, наприклад для застосування оновлених конфігурацій Terraform. Цей програмний інтерфейс має зручний інструментарій, що дозволяє легко налаштовувати КБІ, наприклад завдяки можливості передавати параметри. Серед переваг GitLab CI — проста нативна інтеграція з репозиторіями GitLab, масштабованість та підтримка Docker, що спрощує створення CI/CD-процесів для будь-яких проектів.

3.7 Terraform

Terraform — це інструмент для керування інфраструктурою як кодом (IaC), розроблений HashiCorp. Він дозволяє автоматизувати розгортання, зміни та управління хмарними та локальними ресурсами за допомогою декларативного підходу. Terraform використовує власну мову програмування (HCL — HashiCorp Configuration Language), яка є простою для розуміння та налаштування. Інструмент підтримує різні хмарні провайдери, такі як AWS, Azure, Google Cloud, а також самостійно розміщенні рішення, такі як OpenStack. Завдяки цьому він стає універсальним засобом для оркестрації інфраструктури.

Одна з головних переваг Terraform — це ідемпотентність, тобто можливість багаторазово застосовувати конфігурацію без ризику отримати неконсистентний стан системи. Він також забезпечує версійність, дозволяючи відстежувати зміни в інфраструктурі через систему контрольних версій (наприклад, Git). Terraform має модульну архітектуру, що спрощує повторне використання коду та спільну роботу команд. Крім того, інструмент підтримує паралельне розгортання ресурсів, що значно прискорює процес.

Ще одна ключова перевага — підтримка багатьох провайдерів, що дозволяє керувати гібридною інфраструктурою з єдиного інтерфейсу. Terraform також має механізм планування змін (plan), який показує, які ресурси будуть створені, змінені

або видалені перед фактичним застосуванням. Це зменшує ризик помилок і дозволяє безпечно вносити зміни. Завдяки цим можливостям Terraform став стандартом для автоматизації інфраструктури в сучасних DevOps-процесах.

3.8 IntelliJ IDEA

IntelliJ IDEA — це потужне інтегроване середовище розробки (IDE) від компанії JetBrains, призначене для програмування на мовах Java, Kotlin, Groovy та інших [9]. Воно надає розробникам широкий спектр інструментів для написання, тестування та оптимізації коду. IDE підтримує розумне автодоповнення, аналіз коду в реальному часі та інтеграцію з популярними системами контролю версій, такими як Git.

Одна з головних переваг IDEA — це висока продуктивність завдяки розумним функціям, таким як рефакторинг, пошук помилок і підказки щодо оптимізації коду. IDE також пропонує зручну роботу з фреймворками, наприклад Spring, Hibernate чи Android, що значно прискорює розробку. Інтеграція з інструментами збірки, такими як Maven або Gradle, спрощує управління залежностями. Крім того, IDEA підтримує розширення через плагіни, що дозволяє адаптувати середовище під конкретні потреби.

Ще одна перевага — це крос-платформенність: IDEA працює на Windows, macOS та Linux, забезпечуючи однаковий досвід розробки на різних ОС. Відмінна підтримка тестування (JUnit, TestNG) та налагодження коду робить його ідеальним вибором для професійних розробників. Також IDE інтегрується з хмарними сервісами, такими як Docker і Kubernetes, що важливо для сучасних DevOps-процесів. Завдяки цим можливостям IDEA залишається одним із найпопулярніших середовищ розробки у світі.

Висновки до розділу 3

У цьому розділі було обґрунтовано вибір інструментів та технологій, необхідних для реалізації системи автоматичного масштабування хмарного кластеру кафедри. Кожен із розглянутих засобів був обраний з урахуванням його відповідності вимогам проекту, інтеграційних можливостей та ефективності використання в існуючому середовищі.

Kotlin визначено як основну мову програмування завдяки її лаконічності, сумісності з Java, null-безпеці та можливостям функціонального програмування. Це дозволить розробляти надійний та підтримуваний код, інтегруючи його з іншими компонентами системи. Spring Framework обраний як серверний фреймворк через його модульність, підтримку ін’єкції залежностей, зручні інструменти для роботи з HTTP API та широку екосистему, що включає Spring Boot та Spring Cloud.

Для моніторингу системи обрано Prometheus, оскільки він є ефективним рішенням для збору та аналізу метрик у реальному часі, підтримує потужну мову запитів PromQL та інтегрується з Alertmanager для обробки сповіщень. Alertmanager, у свою чергу, забезпечує гнучке налаштування маршрутизації повідомлень, що дозволяє оперативно реагувати на аномалії в роботі кластеру.

GitLab використовується як платформа для керування кодом, CI/CD-процесами та автоматизації розгортання. Його інтеграція з Terraform дозволяє керувати інфраструктурою як кодом, забезпечуючи ідемпотентність, версійність та безпеку змін. Terraform є оптимальним рішенням для роботи з OpenStack, оскільки дозволяє автоматизувати налаштування ресурсів і підтримує гібридні інфраструктури.

Для розробки програмного забезпечення обрано IntelliJ IDEA — потужне IDE, яке забезпечує високу продуктивність, інтелектуальні інструменти аналізу коду, підтримку Kotlin та Spring, а також інтеграцію з системами збірки та DevOps-інструментами.

У сукупності ці технології утворюють ефективний стек, який задовольняє вимоги проекту щодо автоматизації, масштабованості, безпеки та легкості підтримки. Вибір цих інструментів дозволить реалізувати систему, здатну з легкістю інтегруватися з існуючою інфраструктурою та кодовою базою кафедри та забезпечувати стабільну роботу під час високих навантажень.

4 РОЗРОБКА СИСТЕМИ ДИНАМІЧНОГО МАСШТАБУВАННЯ

Система автоматичного масштабування OpenStack на основі моніторингу навантаження була розроблена задля ефективнішого та незалежного від ручних дій управління інфраструктурою хмари кафедри. Це допоможе підтримувати життєдіяльність серверів інституту під час великого навантаження, напливу користувачів та періодів підвищеної завантаженості без участі спеціалістів, відповідальних за cloud кафедри, що дозволяє уникнути великого часу між інцидентом та реакцією на нього, а також необхідності постійного доступу до налаштувань кластеру хоча б одного з АК.

Така система має змогу як збільшувати інфраструктурні ресурси та кількість працюючих вузлів під час зростання їх використання та навантаження, так і зменшувати їх в разі стабілізації життєвих показників, що збирає СМ. Завдяки перерозподілу обчислювальних потужностей між віртуальними машинами та створення нових, робота кластеру стає більш передбачуваною та відмово стійкою.

4.1 Модель роботи системи

Модель взаємодії компонентів інфраструктури, що відповідають за реалізацію автоматичного масштабування OpenStack на основі моніторингу навантаження, основана на “push” моделі взаємодії мікросервісів між собою задля підвищення ефективності передачі даних та спрощення обробки даних, що були зібрані за допомогою СМ.

Користувачі, а саме DevOps інженери, АК, та інші люди, що відповідають за функціонування хмарної інфраструктури кафедри, імперативно визначають як саме треба обчислювати відповідь для реакції на метрики за певний період часу. Тобто людина, що відповідає за роботу кластеру департаменту інституту, задає математичну функцію, яка буде обчислюватись для отримання результату у вигляді

булевого значення – true або false. Для цього також треба обрати які джерела життєвих показників та які саме життєві показники використовувати в якості параметрів для заданої функції. До того ж, користувач встановлює певний період часу, протягом якого має виконуватись та чи інша умова. Це дає можливість більш гнучко налаштовувати ці правила й запобігати утворенню хибнопозитивних результатів, що можуть перешкоджати правильній роботі системи автоматичного масштабування в разі короткочасних та незначних збільшень в навантаженні.

Після визначення функції обчислення налаштовується отримувач повідомлення про позитивний результат відпрацювання функції. Тобто коли задана попередньо формула дає сигнал про виявлення неочікуваних життєвих показників з вузлів інфраструктури, спеціальний підмодуль у вигляді Prometheus Alertmanager в моніторинговій системі реагує на це, відправляючи повідомлення на попередньо сконфігурований пункт доставки, що в подальшому має обробити це повідомлення, або допоможе сповістити АК про виявлені аномалії. Це може бути бот в месенджері, електронна пошта, Webhook тощо.

В нашому випадку використовується Webhook отримувач на стороні мікросервісного додатку, що відповідає за автоматизоване масштабування інфраструктури. При отриманні повідомлення, модуль динамічного розподілу ресурсів викликає за допомогою прикладного програмного інтерфейсу (API) конвеєр безперервної інтеграції, що запускає процес змін конфігурації інфраструктури. Цей конвеєр в свою чергу вносить зміни в репозиторій системи контролю версій та накладає нові параметри на кластер кафедри за допомогою скрипта з відповідними командами.

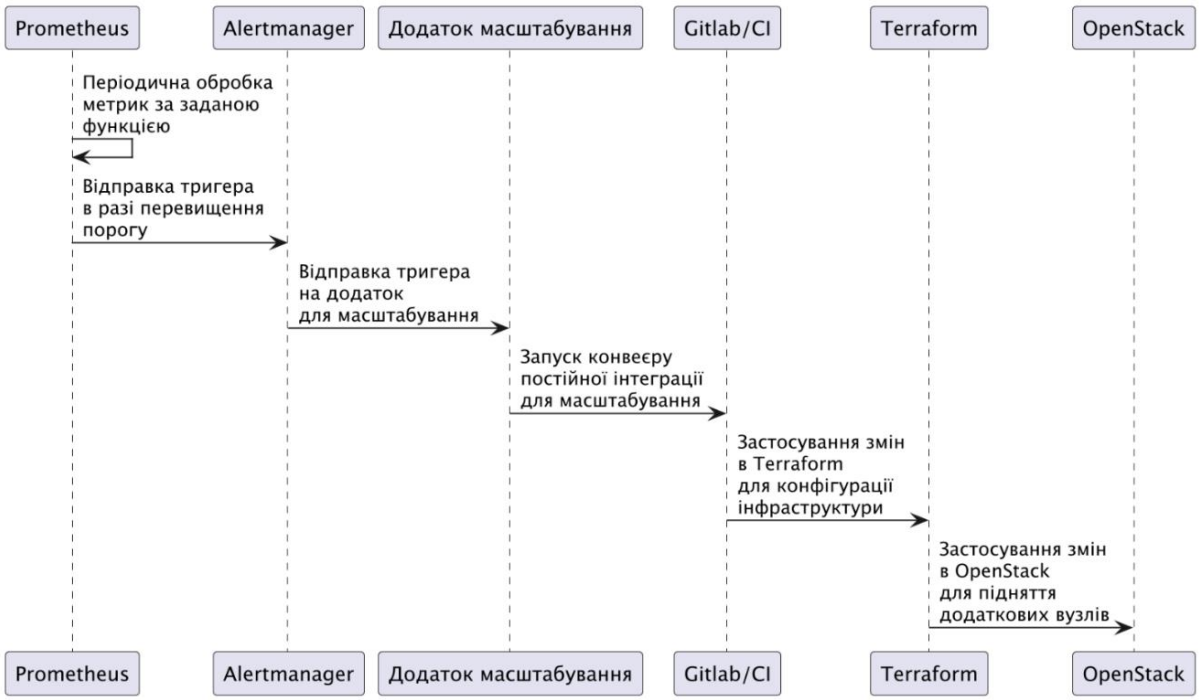


Рисунок 4.1 – UML діаграма роботи системи автоматичного масштабування

Для відпрацювання зворотного процесу та зменшення кількості виділених віртуальних машин використовується той же самий шлях зміни конфігурацій, від моніторингової системи до інструментів постійної інтеграції та постійної доставки. До того ж, для цього зазвичай не потрібно робити додаткові налаштування та якимось чином ускладнювати систему, оскільки модуль відслідковування життєвих показників кластеру інституту має функцію автоматичного встановлення нормального стану після того, як система відновиться і функція перестане давати позитивний результат протягом певного часу. Під час встановлення нормального стану СМ автоматично відправить повідомлення про нормалізацію монітору та стану життєвих показників на сконфігурованих попередньо отримувачів, що й запустить зворотній процес на стороні КБІ.

4.2 Процес розробки системи

Процес розробки системи динамічного масштабування інфраструктури OpenStack на основі моніторингу навантаження є складним багатокроковим процесом, під час якого необхідно поєднати усі необхідні взаємодіючі між собою модулі, налаштувати їх коректну інтеграцію та забезпечити правильне відпрацювання всього циклу роботи цієї системи. Для цього програмне проектування поділено на декілька етапів, кожний з яких відповідає за певну частину життєвого циклу функціонування модулю.

4.2.1 Налаштування системи моніторингу

Першим кроком в розробці була конфігурація Prometheus та Prometheus Alertmanager, що дозволяє в подальшому відслідковувати метрики та поведінку, на які слід орієнтуватись для підтримки інфраструктури кластеру. Це дозволить отримувати сповіщення про аномалії в життєвих показниках, що були виявлені за допомогою СМ, на попередньо налаштованих обробниках, таких як Telegram чат, а також через Webhook, щоб мати можливість в подальшому обробляти його серверним додатком.

Спочатку необхідно підібрати метрики, які ми хочемо відслідковувати та задати їм відповідну формулу. В нашому випадку треба дивитись на показники використання ресурсів доступних процесорів та встановити нормальні рамки для цих метрик. Експериментальним шляхом було підібрано, що оптимальним значенням для цієї формули є використання 50% віртуальних процесорних ресурсів і що така умова має виконуватись щонайменше 10 хвилин, перед тим, як сповіщати про це отримувачів. Для задання такої формули із додатковими полями з інформацією було використано інструменти YAML конфігурації Prometheus (рис. 4.2).

```yaml
- name: Workers Load Alerts
  rules:
    - alert: CpuUsageHighContinuously
      expr: (sum(openstack_nova_vcpus_used) / sum(openstack_nova_vcpus) * 100) > 50
      for: 10m
      labels:
        severity: warning
      annotations:
        summary: "OpenStack workers have high CPU usage continuously"
        description: "More than 50% of CPU used by OpenStack workers (value={{ $value }})."

```

Рисунок 4.2 – YAML конфігурація формули Prometheus

Результати такої конфігурації можна побачити в візуалізованих за допомогою Grafana даних з Prometheus, як зображено на рисунку 4.3.

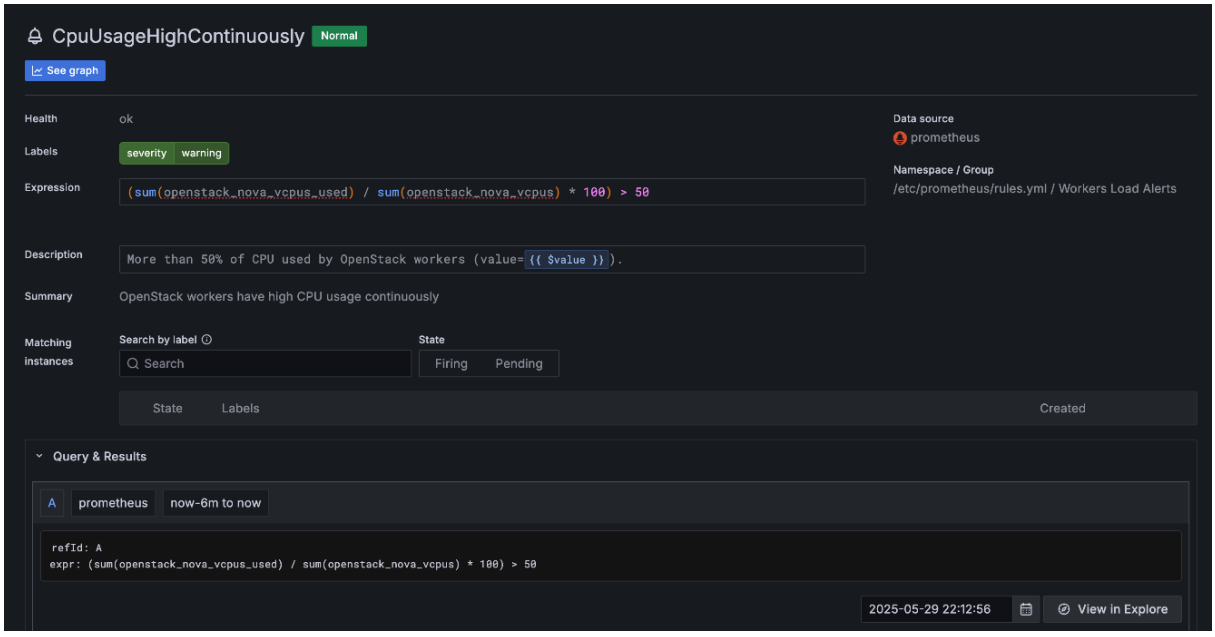


Рисунок 4.3 – Правило повідомлення про підвищене використання CPU в інтерфейсі Grafana

Після цього необхідно було визначити отримувачів повідомлень про аномалії із заданою формулою. А саме, налаштувати параметри Prometheus Alertmanager таким чином, щоб при відпрацювання формули про підвищене навантаження на CPU протягом десяти хвилин, відправлявся Webhook на задану IP-адресу та мережевий порт, задля подальшої його обробки серверним додатком та запуску

процесу масштабування. Для цього також було використано засоби YAML конфігурації Prometheus Alertmanager. Для даного отримувача була встановлена спеціальна фільтрація повідомлень про аномалії в життєвих показниках за назвою, щоб модулю динамічного масштабування відправлялись тільки ті сповіщення, на які потрібно реагувати масштабуванням.

4.2.2 Створення GitLab CI Pipeline

Наступним кроком розробки було необхідно створити та коректно налаштувати GitLab CI Pipeline.

Оскільки конфігурація інфраструктури для нашого кластеру зберігається у Git-репозиторії в самостійно розміщеному модулі GitLab, нам необхідно змінювати налаштування саме в ньому, щоб тим самим провокувати запуск процесу масштабування інфраструктури, а також зберігати оновлену конфігурацію для узгодженості системи.

Для цього спочатку було встановлено правила, за якими цей КБІ буде запускатись, оскільки в іншому випадку процес масштабування буде запускатись при кожній зміні в репозиторії, що є некоректною поведінкою. Для цього ми виставляємо правило про те, що ця CI Job має відпрацьовувати лише коли вона запущена вручну, тобто через прикладний програмний інтерфейс або веб-інтерфейс GitLab, а також що серед параметрів для неї є визначений користувачем параметр, що відповідає за напрямок масштабування (вгору або вниз).

Після визначення правила було розроблено спеціальний bash скрипт, що отримує поточні значення кількості вузлів з Git-репозиторію, на основі параметру про напрямок масштабування визначає нову кількість вузлів, після чого він вносить ці зміни в відповідний файл, робить git commit та відправляє ці зміни на GitLab за допомогою git push. Для цих маніпуляцій також необхідно на рівні налаштувань GitLab CI визначити спеціальні змінні, що будуть зберігати токени доступу до git проекту, оскільки це потрібно для можливості вносити зміни в цей репозиторій.

4.2.3 Розробка серверного додатку

Останнім кроком розробки була розробка програмного модуля, що відповідає за отримання сповіщення, його обробку та ініціалізацію змін в налаштуваннях інфраструктури.

Цей модуль має вміти обробляти Webhook, що представляє з себе HTTP POST запит, отримувати дані, що передані всередині цього Webhook, й в подальшому використовувати їх.

Для створення проекту було використано інструменти Spring Framework, що дозволяють легко та швидко створювати серверні додатки на мовах програмування, що засновані на JVM, в нашому випадку – Kotlin. Першим кроком було створення моделі повідомлення, що відповідала моделі тіла HTTP POST запиту, що надходить з Prometheus Alertmanager. Для цього було ретельно опрацьовано документацію CM. Після створення моделі, було розроблено Spring REST Controller в додатку, що надає зручні інструменти для отримання та обробки HTTP запитів, і створено метод класу, що відповідає саме за обробку необхідного повідомлення.

Після розробки REST Controller важливою частиною була саме інтеграція додатку із прикладним програмним інтерфейсом GitLab CI, що надає можливість запускати КБІ за допомогою HTTP запитів. Для реалізації цього всередині нашого програмного модулю було створено окремий файл із конфігураціями, необхідними для використання API, а саме з даними про провайдера GitLab, що в нашому випадку відрізняється від стандартного “gitlab.com”, з даними про ID проекту та токеном доступу до репозиторію. В початковому коді ці дані замінені заповнювачами, в подальшому вони мають підмінятись реальними даними під час деплою додатку в інфраструктуру кафедрального кластеру.

Ці дані були використані всередині додатку за допомогою спеціальних моделей властивостей, що збирають дані з конфігураційних файлів та агрегують їх у відповідно визначені класи та об’єкти в програмному коді.

Для подальшої інтеграції з GitLab зі сторони додатку, було розроблено клас-адаптер, що збирав запит до API конвеєрів постійної інтеграції у відповідності до вимог та необхідної моделі запиту. Для цього також були використані інструменти Spring Framework, що дають можливість з легкістю будувати запит кодом, використовувати різні формати, в тому числі формат Form для передачі таких значень як параметр напрямку масштабування та токену доступу, а також робити будь які запити на сторонні системи, використовуючи вбудований клас RestTemplate, що значно спрощує роботу з HTTP. В цей адаптер було автоматично впроваджено об’єкт властивостей, які є необхідними для коректної побудови запиту, за допомогою парадигм інверсії контролю та ін’єкції залежностей, що є одними з базових в програмуванні за допомогою Spring Framework.

Діаграму класів кінцевої версії розробленого додатку можна побачити на рисунку 4.4.

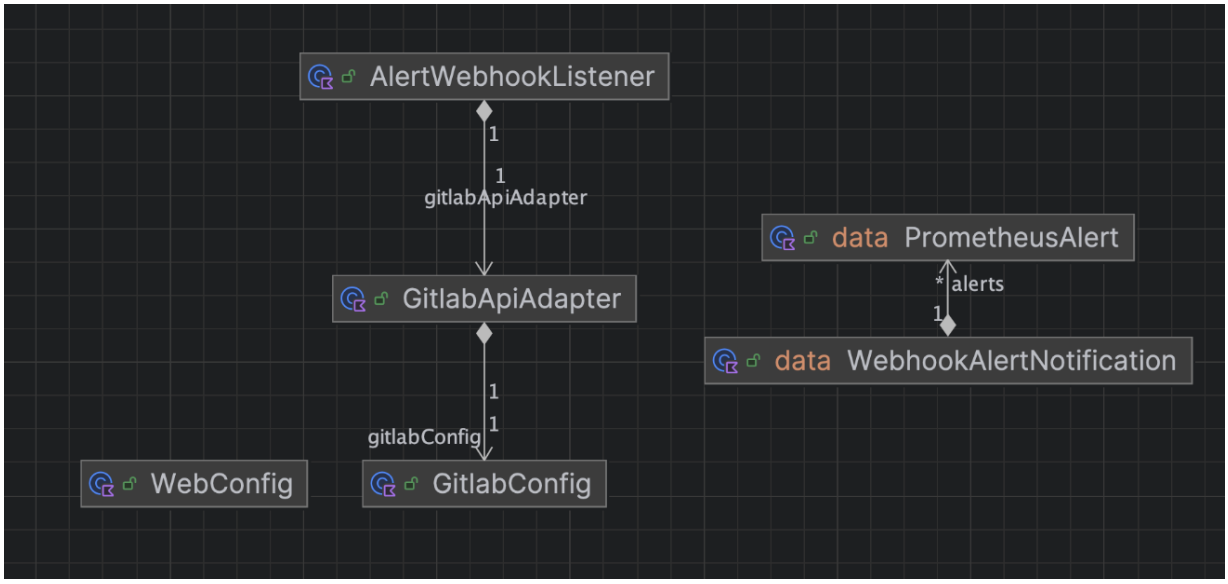


Рисунок 4.4 – Діаграма класів серверного модуля динамічного масштабування

Завдяки використанню фреймворку Spring, розроблений програмний модуль не потребує ручного збирання й компіляції, що дозволяє легко розширювати його й підтримувати працездатність після додавання нових компонентів, класів та

файлів. Це дозволяє в майбутньому розширювати додаток новими адаптерами, інструкціями та імплементовувати більш складну логіку обробки за потреби.

Весь додаток для подальшого використання збирається в Docker Image, для можливості його розгортання як локально, наприклад для тестування та детального налаштування в безпечних умовах, так і на базі хмар, в тому числі хмари кафедри.

Висновки до розділу 4

У даному розділі було розроблено систему динамічного масштабування OpenStack, яка дозволяє автоматично керувати ресурсами хмарної інфраструктури кафедри на основі моніторингу навантаження. Система спрощує управління кластером, усуваючи необхідність постійного втручання адміністраторів, і забезпечує стабільну роботу серверів під час пікових навантажень.

Основою функціонування системи є модель взаємодії компонентів, що базується на push-механізмі передачі даних між мікросервісами. Користувачі, такі як DevOps-інженери, мають змогу гнучко налаштовувати правила масштабування, задаючи умови, за яких система збільшує або зменшує кількість вузлів. Для цього використовується моніторинг метрик за допомогою Prometheus, обробка сповіщень через Alertmanager та подальша інтеграція з GitLab CI для автоматичного внесення змін у конфігурацію інфраструктури.

Процес розробки включав налаштування системи моніторингу, створення GitLab CI Pipeline для управління інфраструктурою як кодом (IaC) та розробку серверного додатку на основі Spring Framework, який обробляє події масштабування. В результаті було створено модуль, що забезпечує автономне масштабування ресурсів, підвищуючи відмовостійкість та ефективність хмарної інфраструктури.

Запропоноване рішення дозволяє оптимізувати використання обчислювальних потужностей, зменшуючи витрати на обслуговування та

забезпечуючи стабільну роботу сервісів навіть у періоди підвищеного навантаження.

5 ТЕСТУВАННЯ СИСТЕМИ

Для переконання в працездатності системи, після розробки було проведено її тестування. Для цього було створено GitLab репозиторій з необхідними налаштуваннями та файлами для його роботи, такими як файл с КБІ та файл конфігурації інфраструктури «terraform.tfvars», ідентичний тому, що використовується в реальній конфігурації кластеру. Також було локально розгорнуто розроблений модуль динамічного масштабування, із використанням відповідних налаштувань, що надають доступ до приватного GitLab репозиторію.

Коли система була в готовому стані, було просимульовано відпрацювання сповіщення про аномально високе використання ресурсів процесору протягом десяти хвилин.

В результаті очікується збільшення кількості робочих вузлів в конфігурації OpenStack, сконфігурованої за допомогою Terraform.

Як можемо побачити на рисунку 5.1, розроблений серверний додаток отримав сповіщення, обробив його, за статусом повідомлення обрав напрямок масштабування та, використовуючи GitLab API, надіслав запит на запуск конвеєру безперервної інтеграції.

Рисунок 5.1 – Журнал обробки сповіщення додатком

Після обробки сповіщення додатком запускається КБІ на стороні GitLab, зі статусом «triggered», що означає, що CI Pipeline був запущений за допомогою програмного інтерфейсу, як ми можемо побачити на рисунку 5.2.

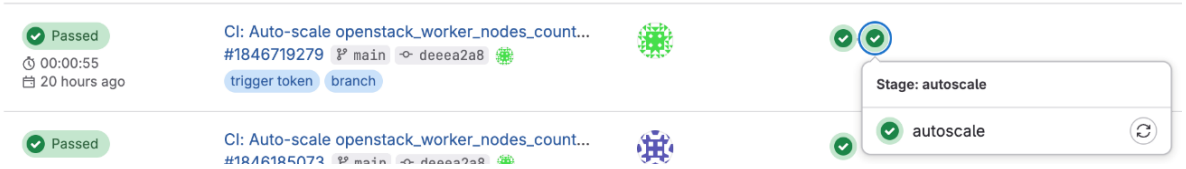


Рисунок 5.2 – Конвеєр безперервної інтеграції в списку GitLab CI Pipelines

В самому КБІ виконується bash скрипт, що відповідає за заміну значень в файлах конфігурації в залежності від переданих в нього параметрів, та виконання “git push” до головної гілки репозиторію конфігурацій, що ми можемо побачити на рисунку 5.3.

```
$ TFVARS_FILE="resources/terraform.tfvars"
$ VARIABLE_NAME="openstack_worker_nodes_count"
$ echo "Executing autoscale job..."
Executing autoscale job...
$ if [ ! -f "${TFVARS_FILE}" ]; then # collapsed multi-line command
$ CURRENT_VALUE=$(grep -E "^\s*${VARIABLE_NAME}\s*=" "${TFVARS_FILE}" | awk -F'=' '{print $2}' | tr -d ' "')
$ if [ -z "${CURRENT_VALUE}" ] || ! [[ "${CURRENT_VALUE}" =~ ^[0-9]+$ ]]; then # collapsed multi-line command
$ if [ "$SCALE_DIRECTION" == "up" ]; then # collapsed multi-line command
$ sed -i -E "s/^(\s*${VARIABLE_NAME}\s*=\s*)[0-9]+/\1${NEW_VALUE}/" "${TFVARS_FILE}"
$ if git diff --quiet "${TFVARS_FILE}"; then # collapsed multi-line command
Changes detected in resources/terraform.tfvars. Committing and pushing...
[main bdc73c8] CI: Auto-scale openstack_worker_nodes_count to 24
 1 file changed, 1 insertion(+), 1 deletion(-)
To https://gitlab.com/DmitryOnopriienkoTV12/ci-test.git
   deeea2a..bdc73c8  HEAD -> main
Changes pushed to main.
```

Рисунок 5.3 – Журнал виконання команд всередині конвеєру безперервної інтеграції на стороні GitLab

Кінцевим результатом роботи всієї системи стало збільшення кількості робочих вузлів OpenStack в конфігурації інфраструктури, заданої за допомогою Terraform. Результат можна побачити на рисунку 5.4.

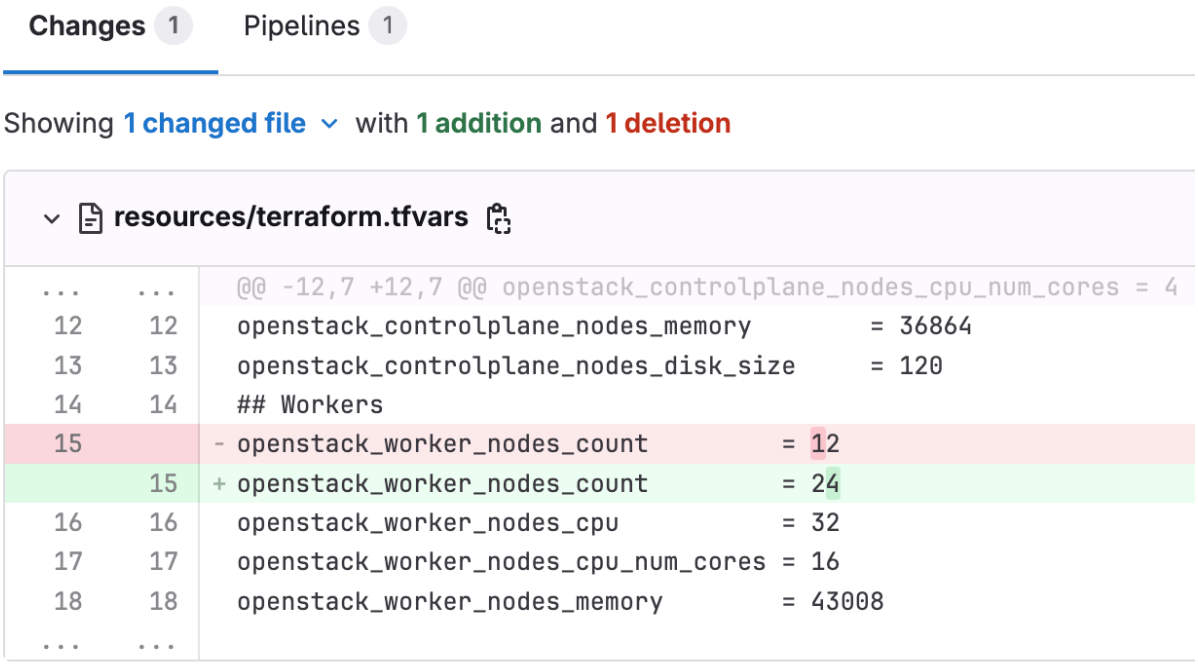


Рисунок 5.4 – Внесені зміни в конфігурацію інфраструктури за допомогою КБІ

В результаті бачимо що система повністю відпрацювала коректно, усі інтеграції відбулись за очікуваннями, без збоїв і був отриманий правильний результат.

В зворотному напрямку система також була протестована аналогічно, але з використанням статусу повідомлення “resolved” і очікуваним зменшенням кількості робочих вузлів кластеру. Під час тестування був отриманий відповідний результат.

Висновки до розділу 5

Проведене тестування системи підтвердило її працездатність та ефективність у динамічному масштабуванні інфраструктури. Система коректно обробляє сповіщення про аномальне навантаження, запускає відповідний GitLab CI Pipeline, що вносить необхідні зміни в конфігурацію інфраструктури OpenStack, заданою через Terraform. Успішне збільшення та зменшення кількості робочих вузлів OpenStack під час тестових сценаріїв довело надійність розробленого рішення. Всі

компоненти системи взаємодіють без збоїв, що підтверджує відповідність системи поставленим критеріям та готовність до використання в реальних умовах на кафедральному кластері.

6 РОЗГОРТАННЯ ТА ПІДТРИМКА СИСТЕМИ

Важливою частиною проекту та життєвого циклу системи є розгортання та підтримка, а також розширення вже існуючого функціоналу. Це аспект, який необхідно враховувати при розробці системи, оскільки її подальше використання буде потребувати додавання нових функцій, нових правил, можливостей тощо. Тож систему треба робити доволі прямолінійною, з чистими та зрозумілими зв'язками, без зав'язки на якусь конкретну конструкцію, щоб давати змогу в майбутньому перевикористати її, замінити, або додати нову.

6.1 Розгортання системи

Необхідним етапом для початку використання системи є її розгортання. Виконання цього процесу коректно є важливим критерієм для правильної роботи додатку та усіх інтеграцій.

Розгортання від початку до кінця включає в себе декілька етапів:

1. Конфігурація СМ;
2. Конфігурація конвеєрів безперервної інтеграції GitLab та встановлення параметрів на рівні репозиторію;
3. Розгортання серверного додатку;
4. Налаштування Prometheus Alertmanager.

Першим етапом для коректного розгортання системи цілком є перевірка наявності налаштувань моніторів в системі Prometheus, що задаються за допомогою YAML конфігурацій. Приклад необхідної конфігурації міститься в коді проекту. Така конфігурація додається в GitLab репозиторій кафедри, що відповідає за налаштування моніторингу та застосовує ці налаштування до існуючої СМ. Перевірити результат можна у веб-інтерфейсі кафедральної Grafana, що візуалізує дані та налаштування з Prometheus. Там ми маємо побачити правило про повідомлення в разі відпрацювання заданої формули.

Наступним кроком розгортання є перевірка наявності або створення конвеєру безперервної інтеграції в кафедральному GitLab проекті OPENSTACK, в разі відсутності такої Job. Вона має використовувати наявні в проекті задані параметри, такі як токени доступу. Тому важливо перевірити їх наявність і в разі відсутності – створити. Після цього важливо перевірити шляхи до файлів, що використовуються у bash скрипті КБІ, в нашому випадку – шлях до файлу «terraform.tfvars», що відповідає за конфігурації ресурсів інфраструктури. Приклад коду для CI Pipeline Job наявний в коді дипломного проекту.

Після цього йде етап розгортання серверного додатку. Для цього необхідно створити віртуальну машину всередині кафедрального кластеру із доступом з приватної мережі. Оскільки цей додаток працює цілком всередині кластеру й не має отримувати запити ззовні, а тільки зсередини прихованої мережі кластеру, в якій знаходиться Prometheus, то немає сенсу відкривати доступ до неї з публічної мережі. Це також дозволяє зберігати високий рівень безпеки для кластеру.

Після створення віртуальної машини, необхідно підключитись до неї та створити файл з конфігураціями для серверного додатку, що буде містити дані про провайдера GitLab, ID необхідного проекту, де буде запускатись CI Pipeline, а також токен доступу, щоб мати змогу ініціалізувати запуск КБІ за допомогою прикладного програмного інтерфейсу. Приклад такого файлу можна побачити на рисунку 6.1.

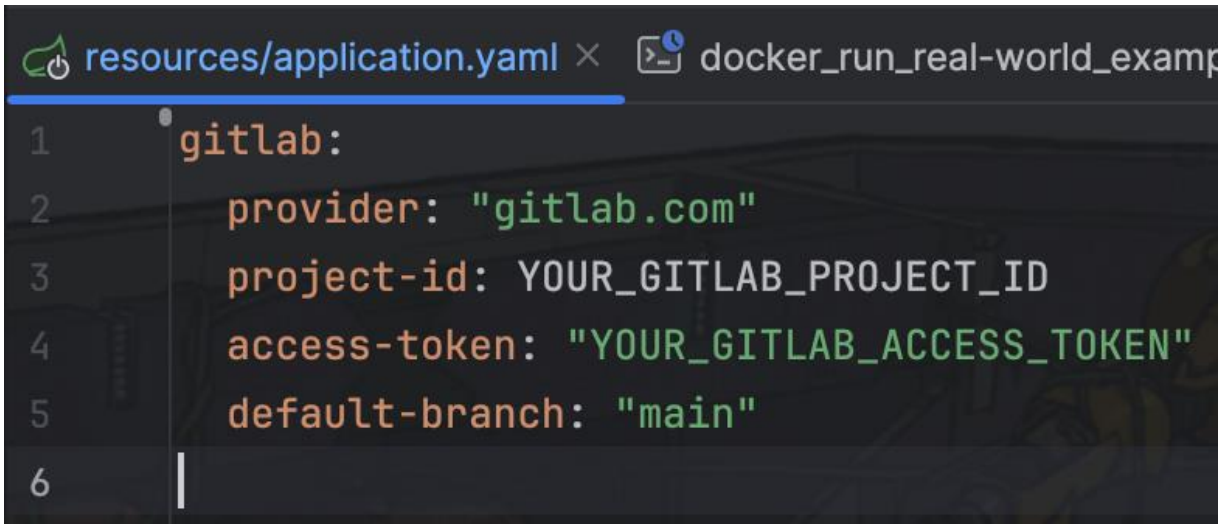


Рисунок 6.1 – Приклад файлу конфігурацій для серверного додатку

Після створення конфігураційного файлу треба стягнути з image репозиторію кафедри та запустити його, використовуючи команду “docker run” із спеціальними параметрами. Ці параметри відповідають за завантаження файлу конфігурацій всередину контейнера та його подальшого використання в додатку в якості значень, що будуть затирати собою початкові. Це дасть змогу контейнеру використовувати GitLab API, оскільки без цього запити будуть створюватись некоректно. Приклад відповідної команди є у коді проекту.

Останнім кроком є налаштування отримувача сповіщень Prometheus Alertmanager таким чином, щоб він відфільтровував усі повідомлення, що нам не підходять, та надсилав лише потрібні через Webhook. Для цього нам треба дізнатись IP-адресу нашої віртуальної машини та проставити її в якості URL для отримувача, з портом 8080, що використовується нашим Spring-додатком, і URI адресою, що в нашому випадку є “/alert/Prometheus/high-cpu”. А також необхідно виставити фільтрацію за назвою повідомлення, що задається в Prometheus.

Після виконаних кроків система буде готова для використання та буде реагувати на відповідні повідомлення та аномалії в життєвих показниках кафедрального кластеру.

6.2 Підтримка та розширення системи

В подальшому система може з легкістю бути розширена за допомогою додавання нового функціоналу, розширення та переналаштування існуючого.

Це може бути досягнуто, наприклад, за допомогою додавання нових правил повідомлення про аномалії в життєвих показниках кафедрального кластеру. Наявна можливість додати такі формули, що будуть дивитись не на використання процесору, а на інші показники метрик серверу, наприклад заповнення оперативної пам’яті кластеру, що також впливає на можливість серверів департамента інституту обробляти запити та проводити обчислювальні операції.

Окрім того, можна переналаштовувати вже існуючі правила, тобто якщо при подальшому масштабуванні та розширенні загальних ресурсів кластеру, а також зростання середнього навантаження, буде потреба в зміні формули обчислення аномального стану системи та часу, за який ця формула має давати позитивний результат – це можна змінити централізовано через репозиторій GitLab з відповідними конфігураціями CM.

Також є можливість додати додаткову логіку в серверний додаток, що обробляє повідомлення про аномалії, оскільки він за структурою доволі простий, і технології, що використані в ньому, мають доволі низький поріг входу для реалізації не складного функціоналу.

Окрім наведених вище можливостей, в майбутньому також можна змінювати поведінку КБІ задля досягнення більш точкових результатів, зміни інших параметрів і створення більш гнучкого налаштування логіки скрипту.

Висновки до розділу 6

У даному розділі було детально розглянуто процес розгортання, який включає налаштування моніторингу, конфігурацію конвеєрів CI/CD, розгортання серверного додатку та інтеграцію з Prometheus Alertmanager. Кожен із цих кроків є

невід’ємною частиною загальної системи, і їх правильне виконання забезпечує коректну реакцію на аномалії в роботі кластеру.

Окрему увагу приділено підтримці та можливостям розширення функціоналу. Система спроектована таким чином, щоб у майбутньому можна було легко додавати нові правила моніторингу, змінювати існуючі параметри або вдосконалювати логіку роботи серверного додатку. Гнучкість архітектури та використання сучасних інструментів розгортання дозволяють адаптувати систему до нових вимог без значних зусиль.

Таким чином, запропонований підхід до розгортання та підтримки забезпечує не лише стабільність системи на поточному етапі, але й створює основу для її розвитку в майбутньому. Це дозволить ефективно реагувати на зміни в навантаженні та інфраструктурі, забезпечуючи безперебійну роботу кластеру та оперативне усунення потенційних проблем.

ВИСНОВКИ

У даній дипломній роботі було розроблено програмний модуль для динамічного масштабування OpenStack-інфраструктури на основі моніторингу навантаження. Метою роботи була автоматизація управління ресурсами кафедрального кластеру, що дозволить запобігати критичним навантаженням, зменшити залежність від ручних операцій та підвищити стабільність системи. Реалізація цієї мети була досягнута завдяки інтеграції трьох основних компонентів: системи моніторингу Prometheus, модуля конфігурації інфраструктури Terraform та серверного додатку, який виступає посередником між цими підсистемами.

Робота почалася з аналізу предметної області, де було виявлено, що існуючі рішення для автоматичного масштабування, такі як AWS Auto Scaling Groups або Kubernetes Horizontal Pod Autoscaling, не підходять для специфіки кафедрального кластеру через його унікальну архітектуру та використання OpenStack. Це обумовило необхідність розробки власного модуля, який би враховував особливості наявної інфраструктури. Важливим аспектом стало використання "push" моделі взаємодії між компонентами, що дозволило спростити обробку даних та підвищити ефективність системи.

Для реалізації проекту було обрано сучасний стек технологій: мову програмування Kotlin, фреймворк Spring для серверного додатку, Prometheus для моніторингу, Terraform для управління інфраструктурою та GitLab CI для автоматизації процесів. Такі інструменти були обрані через їхню сумісність, гнучкість та зручність інтеграції, що дозволило створити надійну та масштабовану систему. Ключовою перевагою використаних технологій є їхня здатність працювати в умовах обмежених ресурсів, що критично важливо для кафедрального середовища.

Розроблений модуль пройшов тестування, яке підтвердило його працездатність. Система коректно реагує на аномальне навантаження, запускаючи процес масштабування через GitLab CI та вносячи необхідні зміни в конфігурацію

інфраструктури. Тестування також показало, що система здатна повертати попередній стан після стабілізації навантаження, що доводить її ефективність у реальних умовах.

Важливим аспектом роботи є можливість подальшого розширення функціоналу. Наприклад, можна додати нові правила моніторингу, змінити параметри масштабування або вдосконалити логіку серверного додатку. Гнучкість архітектури дозволяє легко адаптувати систему до нових вимог, що робить її майбутнє впровадження більш перспективним.

У цілому, дипломна робота демонструє, що розроблений модуль є ефективним рішенням для автоматизації масштабування OpenStack-інфраструктури. Він не лише підвищує стабільність роботи кластеру, але й зменшує навантаження на адміністраторів, дозволяючи їм зосередитися на інших важливих завданнях. Реалізація проекту відкриває нові можливості для оптимізації ресурсів кафедри та забезпечує основу для подальшого вдосконалення системи в майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Omar Khedher - Mastering OpenStack, 3rd Edition: Implement the latest techniques for designing and deploying an operational, production-ready private cloud. – 2024.
2. Kotlin Docs - https://kotlinlang.org/docs/home.html.
3. Prometheus Documentation - https://prometheus.io/docs/introduction/overview/.
4. Terraform Documentation - https://developer.hashicorp.com/terraform/docs.
5. Git Documentation - https://git-scm.com/doc.
6. GitLab Documentation - https://docs.gitlab.com/.
7. Spring Framework Documentation - https://docs.spring.io/spring-framework/reference/index.html.
8. Alex Caldwell - Java Programming For Cloud Computing With Azure SDK And OpenStack SDK: A Hand's-On Beginner's Guide To Leveraging Azure SDK And OpenStack SDK For Robust. – 2024.
9. IntelliJ IDEA Docs – https://www.jetbrains.com/help/idea/discover-intellij-idea.html.

ДОДАТОК А Лістинг розробленої системи

AlertWebhookListener.kt

```
package com.onopriienko.autoscalerapp.controller

import com.onopriienko.autoscalerapp.enums.AlertStatus
import com.onopriienko.autoscalerapp.enums.ScaleDirection
import com.onopriienko.autoscalerapp.model.WebhookAlertNotification
import com.onopriienko.autoscalerapp.service.GitlabApiAdapter
import org.slf4j.LoggerFactory
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/alert/prometheus")
class AlertWebhookListener(
    private val gitlabApiAdapter: GitlabApiAdapter,
) {

    @PostMapping("/high-cpu")
    fun handlePrometheusAlert(@RequestBody alert: WebhookAlertNotification) {
        LOG.info("Received alert: {}", alert)
        val alertStatus = runCatching {
            val prometheusAlert = alert.alerts?.first()
            enumValueOf<AlertStatus>(requireNotNull(prometheusAlert?.status).uppercase())
        }.getOrElse {
            LOG.error("Error while handling webhook alert: {}", alert, it)
            return
        }
        when (alertStatus) {
            AlertStatus.FIRING                                                    ->
gitlabApiAdapter.triggerScalingCiPipeline(ScaleDirection.UP)
            AlertStatus.RESOLVED                                                  ->
gitlabApiAdapter.triggerScalingCiPipeline(ScaleDirection.DOWN)
        }
    }
```

```
    companion object {
        private val LOG = LoggerFactory.getLogger(AlertWebhookListener::class.java)
    }
}
```

GitlabApiAdapter.kt

```
package com.onopriienko.autoscalerapp.service

import com.onopriienko.autoscalerapp.config.GitlabConfig
import com.onopriienko.autoscalerapp.enums.ScaleDirection
import org.slf4j.LoggerFactory
import org.springframework.http.HttpEntity
import org.springframework.http.HttpHeaders
import org.springframework.http.MediaType
import org.springframework.stereotype.Component
import org.springframework.util.LinkedMultiValueMap
import org.springframework.util.MultiValueMap
import org.springframework.web.client.RestTemplate
import org.springframework.web.util.UriComponentsBuilder

@Component
class GitlabApiAdapter(
    private val restTemplate: RestTemplate,
    private val gitlabConfig: GitlabConfig,
) {
    fun triggerScalingCiPipeline(scaleDirection: ScaleDirection) {
        val formattedUri = String.format(CI_JOB_URL_TEMPLATE, gitlabConfig.provider,
gitlabConfig.projectId)
        val uri = UriComponentsBuilder.fromUriString(formattedUri)
            .build().toUri()

        val formData: MultiValueMap<String, String> = LinkedMultiValueMap()
        formData.add(TOKEN_FORM_NAME, gitlabConfig.accessToken)
        formData.add(REF_FORM_NAME, gitlabConfig.defaultBranch)
        formData.add(SCALE_DIRECTION_VARIABLE_FORM_NAME, scaleDirection.value)

        val headers = HttpHeaders()
        headers.contentType = MediaType.MULTIPART_FORM_DATA
```

```
        val httpEntity = HttpEntity(formData, headers)
        LOG.info("Triggering scaling job (scale direction - {}): {}", scaleDirection.value,
uri)
        val response = restTemplate.postForEntity(
            uri,
            httpEntity,
            String::class.java,
        )
        LOG.info("Result of job triggering: {}", response.body)
    }

    companion object {
        private            const            val            CI_JOB_URL_TEMPLATE            =
"https://%s/api/v4/projects/%s/trigger/pipeline"
        private const val TOKEN_FORM_NAME = "token"
        private const val REF_FORM_NAME = "ref"
        private const val SCALE_DIRECTION_VARIABLE_FORM_NAME = "variables[SCALE_DIRECTION]"

        private val LOG = LoggerFactory.getLogger(GitlabApiAdapter::class.java)
    }
}
```

.gitlab-ci.yml

```
stages:
  - echo
  - autoscale

echo:
  stage: echo
  script:
    echo "Hello World!"

autoscale:
  stage: autoscale
  image: ubuntu:24.04
  before_script:
    - apt-get update && apt-get install -y git
    - apt-get install -y bash coreutils
```

```
    - git config user.email "d.onopriienko.tv12@kpi.ua"
    - git config user.name "ci-bot"
    - git checkout "${CI_DEFAULT_BRANCH}"
    - git pull origin "${CI_DEFAULT_BRANCH}"
  script:
    - TFVARS_FILE="resources/terraform.tfvars"
    - VARIABLE_NAME="openstack_worker_nodes_count"
    - echo "Executing autoscale job..."
    - |
      if [ ! -f "${TFVARS_FILE}" ]; then
        echo "ERROR: ${TFVARS_FILE} not found!"
        exit 1
      fi
    -  CURRENT_VALUE=$(grep  -E  "^\s*${VARIABLE_NAME}\s*="  "${TFVARS_FILE}"  |  awk  -F'='
'{print $2}' | tr -d ' "')
    - |
      if [ -z "${CURRENT_VALUE}" ] || ! [[ "${CURRENT_VALUE}" =~ ^[0-9]+$ ]]; then
        echo "ERROR: Could not find or parse '${VARIABLE_NAME}' in ${TFVARS_FILE}, or it's
not a number."
        echo "Found value: '${CURRENT_VALUE}'"
        exit 1
      fi
    - |
      if [ "$SCALE_DIRECTION" == "up" ]; then
        NEW_VALUE=$((CURRENT_VALUE * 2))
      elif [ "$SCALE_DIRECTION" == "down" ]; then
        NEW_VALUE=$((CURRENT_VALUE / 2))
      else
        echo "Invalid SCALE_DIRECTION value. Use 'up' or 'down'."
        exit 1
      fi
    - sed -i -E "s/^(\s*${VARIABLE_NAME}\s*=\s*)[0-9]+/\1${NEW_VALUE}/" "${TFVARS_FILE}"
    - |
      if git diff --quiet "${TFVARS_FILE}"; then
        echo "No changes to ${TFVARS_FILE}. Value might already be the target or sed failed."
      else
        echo "Changes detected in ${TFVARS_FILE}. Committing and pushing..."
        git add "${TFVARS_FILE}"
        git commit -m "CI: Auto-scale ${VARIABLE_NAME} to ${NEW_VALUE}"
```

```
        git push "https://oauth2:$ACCESS_TOKEN@${CI_SERVER_HOST}/DmitryOnopriienkoTV12/ci-test.git" "HEAD:${CI_DEFAULT_BRANCH}"
        echo "Changes pushed to ${CI_DEFAULT_BRANCH}."
      fi
  rules:
    - if: $CI_PIPELINE_SOURCE == "trigger" && $SCALE_DIRECTION != null
```

prometheus/rules.yaml

```
groups:
  - name: Workers Load Alerts
    rules:
      - alert: CpuUsageHighContinuously
        expr: (sum(openstack_nova_vcpus_used) / sum(openstack_nova_vcpus) * 100) > 50
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "OpenStack workers have high CPU usage continuously"
          description: "More than 50% of CPU used by OpenStack workers (value={{ $value }})."
```

alertmanager.yaml

```
global:

route:
  receiver: telegram-alerts
  group_by: ['alertname', 'severity']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 1h

  routes:
  - receiver: 'scaling-webhook'
    match:
      alertname: 'CpuUsageHighContinuously'

receivers:
```

```
- name: scaling-webhook
  webhook_configs:
  - url: 127.0.0.1:8080/alert/Prometheus/high-cpu
```

ДОДАТОК Б Презентація

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Програмний модуль для динамічного масштабування OpenStack на основі моніторингу навантаження

виконав: студент групи ТВ-12 Онопрієнко Дмитро Олегович

керівник: асистент Гейко Олег Олександрович

2025

Актуальність теми

Метою даної роботи є реалізація автоматизації масштабування існуючої хмарної інфраструктури кафедри в разі надмірного навантаження.

Ця задача є актуальною для кафедрального кластеру, оскільки його підтримка потребує спеціальних знань, навичок та постійного контролю, а ситуації зі зростанням навантаження на університетські сервери в певні періоди часу - дуже поширена. Така система допомагає автоматизувати процеси підтримки життєдіяльності і дозволяє адміністраторам не брати участі в ручному запобіганню відмов кафедрального кластеру.

Постановка задачі

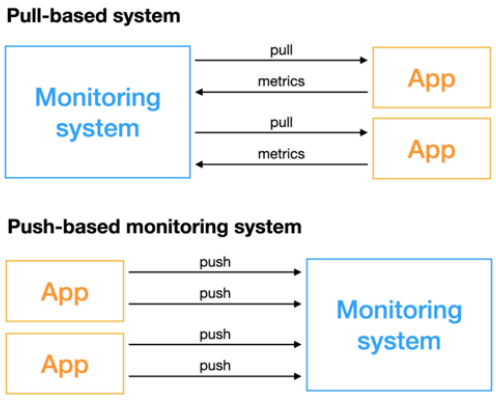
Задача - розробити програмний модуль для автоматизації масштабування вузлів кафедральних серверів на основі даних про навантаження.

Для розв'язання задачі було необхідно:

1. Розробити серверний додаток, що відповідає за прийняття рішень про масштабування
2. Інтегрувати систему моніторинга з додатком
3. Налаштувати правила повідомлень про аномалії в життєвих показниках
4. Сконфігурувати взаємодію додатку з Terraform та Gitlab CI
5. Розробити CI Pipeline, що буде змінювати налаштування інфраструктури по запиту

Аналіз предметної області

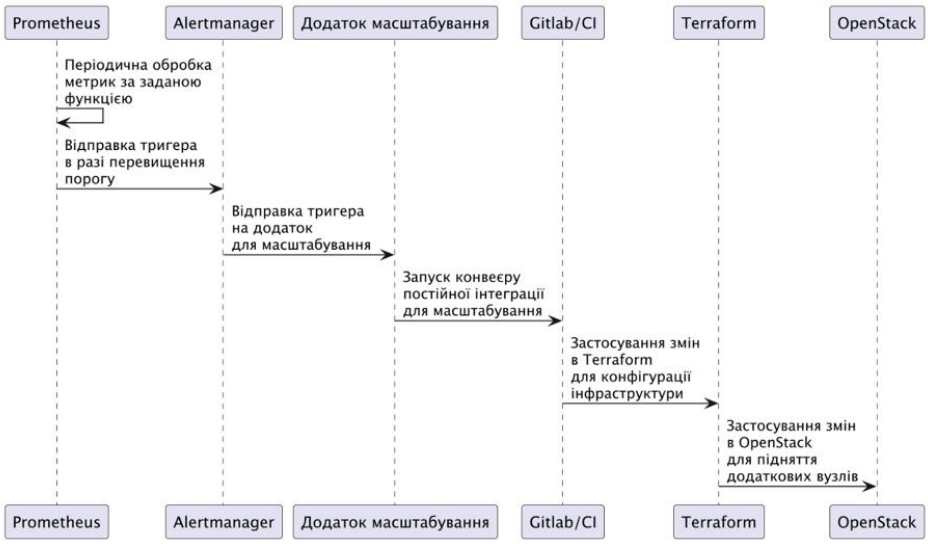
Аналогічні системи мають різну складність їх інтеграції в нашу систему та більша частина з них розроблена для використання в інших Cloud провайдерах, таких як AWS, Google Cloud, Azure. Вони мають різні системи взаємодії з системами збору метрик - за push та pull моделями.



Проєктування системи

Розроблений модуль інтегрується в наявну інфраструктуру і має пов’язувати Prometheus, Gitlab CI та Terraform.
Система має реагувати на повідомлення про довгострокове підвищене навантаження та додавати додаткові вузли в конфігурації інфраструктури, що робиться за допомогою Terraform.
Також, система має реагувати на зменшення навантаження та запускати зворотній процес.
Розроблена інтеграція має не реагувати на короткострокові підвищення навантаження, тільки на тривалі.

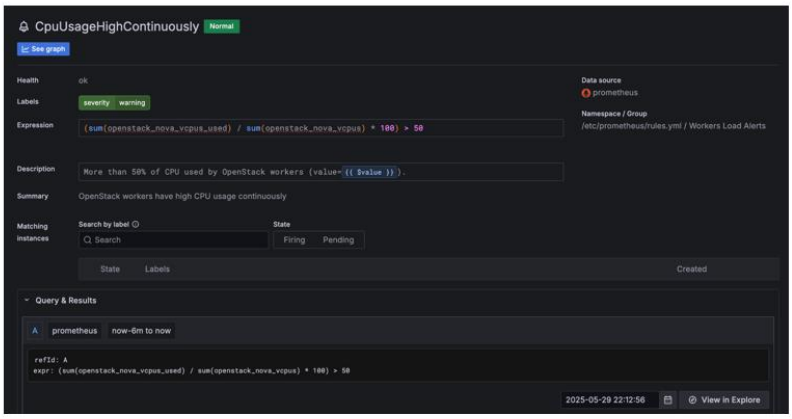
Проєктування системи



Реалізація системи

Першим кроком в розробці було налаштування формули для аналізу метрик та повідомлення про вихід їх з нормального діапазону. Для цього було використано інструменти YAML конфігурації, наявні в Prometheus та Alertmanager.

```yaml
- name: Workers Load Alerts
  rules:
    - alert: CpuUsageHighContinuously
      expr: (sum(openstack_nova_vcpus_used) / sum(openstack_nova_vcpus) * 100) > 50
      for: 10m
      labels:
        severity: warning
      annotations:
        summary: "OpenStack workers have high CPU usage continuously"
        description: "More than 50% of CPU used by OpenStack workers (value={{ $value }})."
```



Реалізація системи

Другий крок імплементації - написання Gitlab CI Pipeline, що буде змінювати конфігурації Terraform в Git репозиторії, відповідальному за налаштування інфраструктури.

Останній крок розробки - реалізація модуля масштабування, що є відповідальним за отримання сповіщень, та реагування на них в залежності від їх наповнення (firing або resolved). Для цього був створений бекенд додаток з використанням Kotlin та фреймворку Spring, що дозволяє легко створювати серверні додатки на мовах програмування, що працюють на JVM.

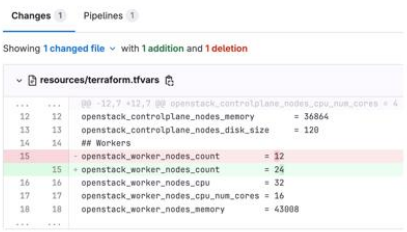
Тестування системи

Було виконано мануальне тестування системи, що показало її повну працездатність та відповідність умовам. Для цього було піднято аналогічну систему та створено окремий репозиторій для симуляції зміни налаштувань конфігурації інфраструктури через Terraform.


```
2025-05-31T19:44:26.314+03:00  INFO 94743 --- [autoscaler-app] [nio-8080-exec-1] c.o.a.controller.AlertWebhookListener     : Received alert:
WebhookAlertNotification(commonLabels=CommonLabels(alertname=CpuUsageHighContinuously, severity=warning), alerts=[PrometheusAlert(status=firing,
generatorURL=http://prometheus.example.com/graph?g0.expr=(sum(openstack_nova_vcpus_used)%20%2F%20sum(openstack_nova_vcpus)%20*%20100)%20%3E%2050,
startsAt=2025-05-27T12:00:00Z)], externalURL=http://alertmanager.example.com)
2025-05-31T19:44:26.321+03:00  INFO 94743 --- [autoscaler-app] [nio-8080-exec-1] c.o.a.service.GitlabApiAdapter          : Triggering scaling job (scale
  direction - up): https://gitlab.com/api/v4/projects/70186669/trigger/pipeline
2025-05-31T19:44:27.449+03:00  INFO 94743 --- [autoscaler-app] [nio-8080-exec-1] c.o.a.service.GitlabApiAdapter          : Result of job triggering:
{"id":1846719279,"iid":70,"project_id":70186669,"sha":"deeea2a8f2025a4f639d95fb7c6dfa49cf6015f6","ref":"main","status":"created","source":"trigger",
"created_at":"2025-05-31T16:44:27.227Z","updated_at":"2025-05-31T16:44:27.227Z","web_url":"https://gitlab
.com/DmitryOnopriienkoTV12/ci-test/-/pipelines/1846719279","before_sha":"0000000000000000000000000000000000000000","tag":false,"yaml_errors":null,
"user":{"id":12684551,"username":"DmitryOnopriienkoTV12","public_email":"","name":"DmitryOnopriienkoTV12","state":"active","locked":false,
"avatar_url":"https://secure.gravatar.com/avatar/2aa59ed14a46e1ff53b38f0a99c17eecd2246a46b50ae186a16bdcda343075fa?s=80\u0026d=identicon",
"web_url":"https://gitlab.com/DmitryOnopriienkoTV12"},"started_at":null,"finished_at":null,"committed_at":null,"duration":null,"queued_duration":null,
"coverage":null,"detailed_status":{"icon":"status_created","text":"Created","label":"created","group":"created","tooltip":"created","has_details":false,
"details_path":"/DmitryOnopriienkoTV12/ci-test/-/pipelines/1846719279","illustration":null,
```


Тестування системи


```
349  $ TFVARS_FILE="resources/terraform.tfvars"
350  $ VARIABLE_NAME="openstack_worker_nodes_count"
351  $ echo "Executing autoscale job..."
352  Executing autoscale job...
353  $ if [ ! -f "${TFVARS_FILE}" ]; then # collapsed multi-line command
354  $ CURRENT_VALUE=$(grep -E "^\s*${VARIABLE_NAME}\s*=" "${TFVARS_FILE}" | awk -F'=' '{print $2}' | tr -d ' "')
355  $ if [ -z "${CURRENT_VALUE}" ] || ! [[ "${CURRENT_VALUE}" =~ ^[0-9]+$ ]]; then # collapsed multi-line command
356  $ if [ "$SCALE_DIRECTION" == "up" ]; then # collapsed multi-line command
357  $ sed -i -E "s/^(\s*${VARIABLE_NAME}\s*=\s*)[0-9]+/\1${NEW_VALUE}/" "${TFVARS_FILE}"
358  $ if git diff --quiet "${TFVARS_FILE}"; then # collapsed multi-line command
359  Changes detected in resources/terraform.tfvars. Committing and pushing...
360  [main bdc73c8] CI: Auto-scale openstack_worker_nodes_count to 24
361   1 file changed, 1 insertion(+), 1 deletion(-)
362  To https://gitlab.com/DmitryOnopriienkoTV12/ci-test.git
363     deeea2a..bdc73c8  HEAD -> main
364  Changes pushed to main.
```



Впровадження та супровід

Була розроблена інструкція з використання системи та її впровадження в інфраструктуру.
Запуск модуля в кафедральному кластері складається з таких кроків:

1. Внесення змін в конфігурацію моніторингової системи із зазначенням правил повідомлень
2. Інтеграція розробленої частини CI Pipeline у наявний кафедральний Gitlab
3. Запуск імплементованого програмного модуля на базі серверів інституту
4. Налаштування на стороні Alertmanager, що дозволяє використовувати бекенд додаток в якості отримувача повідомлення про аномалії в метриках

Висновки

Результатом виконаної роботи є інтегрований в інфраструктуру кафедрального кластеру модуль динамічного масштабування на основі моніторингу навантаження, що автоматично додає в систему нові вузли, коли на серверах інституту виявлено аномальні життєві показники та підвищене використання обчислювальних потужностей.
Для розробки було проаналізовано аналогічні системи, розроблено та протестовано програмний модуль, відповідальний за інтеграції з необхідними частинами системи.
Модуль інтеграції є придатним для використання у кластері інституту та допомагає автоматизувати коригування ресурсів для інфраструктури і її вузлів.
В подальшому ця система може бути розширена, можуть бути додані нові правила для масштабування з можливістю подальшої переконфігурації, а також може бути налаштована більш гнучка поведінка модулю за потребами кафедри.


Національний технічний університет України
«Київський політехнічний інститутімені Ігоря Сікорського»
National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”